AD-A265 641

DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

timated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1993	Professional paper

4. TITLE AND SUBTITLE

AN OBJECT-ORIENTED PARALLEL SIMULATION ENVIRONMENT

5. FUNDING NUMBERS

PR: ECB2
PE: 0602234N
WU: DN30086

6. AUTHOR(S)

L. J. Peterson and P. C-Y Sheu

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Command, Control and Ocean Surveillance Center (NCCOSC)
RDT&E Division
San Diego, CA 92152-5001

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Naval Command, Control and Ocean Surveillance Center (NCCOSC)
RDT&E Division
Block Programs
San Diego, CA 92152-5001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

DTIC
ELECTE
JUN 10 1993
S D
A

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This paper describes an approach to parallel object-oriented simulation. Parallel evaluation of simulation programs is accomplished by compiling objects into sets and production rules so that they can be evaluated with parallel, set-oriented operations which effectively utilize the capacity of parallel processors with minimal communications overhead.

93-12969

93 6 09 050

Published in *Proceedings 1993 Object Oriented Simulated Conference*, January, 1993.

14. SUBJECT TERMS	15. NUMBER OF PAGES
massive parallel processing high performance computing	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS REPORT

21a. NAME OF RESPONSIBLE INDIVIDUAL	21b. TELEPHONE (include Area Code)	21c. OFFICE SYMBOL
L. J. Peterson	(619) 553-4070	Code 421

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1 | 20

AN OBJECT-ORIENTED PARALLEL SIMULATION ENVIRONMENT

Phillip C-Y. Sheu (sheu@caip.rutgers.edu)
Department of Electrical & Computer Engineering
Rutgers University
Piscataway, NJ 08855

Larry J. Peterson (ljp@nosc.mil)
Naval Command, Control and Ocean Surveillance Center
RDT&E Division (NRaD), Code 421
San Diego, CA 92152-5000

Abstract[1] This paper describes an approach to parallel object-oriented simulation. Parallel evaluation of simulation programs is accomplished by compiling objects into sets and production rules so that they can be evaluated with parallel, set-oriented operations which effectively utilize the capacity of parallel processors with minimal communication overhead.

1. Introduction

Since Smalltalk was introduced in the early 80's, it has been generally accepted that object-oriented programming languages can provide a number of attractive features for software development. In an object-oriented system, data can be naturally expressed as a set of objects. This feature is very general and can handle any type of conceptual abstraction and organization. A collection of abstract classes and their associated algorithms (methods), constitute the kind of framework into which particular applications can insert their own specialized code by constructing concrete subclasses that work together. With the method/message protocol, objects can be accessed without knowledge of their internal structures. Classes can be organized into hierarchies so that all methods and attributes of a class can be automatically inherited by all of its subclasses. In addition, the nature of object-oriented representations suggests that an object-oriented simulation program can be executed in parallel.

Unfortunately, despite the nice features described above, the object-oriented methodology has focused on general-purpose programming with no definition of object control, management, and information retrieval. Our research by and large addresses the area called "active object base". Our definition of active objects goes beyond the traditional object-oriented paradigm by including a control component in each object, so that it can always be active, capable of performing state transitions and handling asynchronous events. Hence an *active object base* is defined to be a large collection of objects instrumented with states and asynchronous state transitions. Such a model can be best illustrated by a battlefield, in which all command and control units are active objects, and whose states are extremely sensitive to environmental changes. Our research addresses five major problems associated with active object bases: representation, presentation, management, programming, and execution. This paper reports the representation and execution aspects of an object base, taking simulation as the domain. By representing the control portion of an active object as a production system, parallel evaluation of a simulation program is accomplished by compiling objects onto an interconnected network, so that they can be evaluated with parallel, set-oriented operations which effectively utilize the capacity of parallel processors with minimal communication overhead. This paper is organized into the following sections: Survey of Related Work, The Object Model, Object-Oriented Simulation, Rule Processing, Incremental Network Compilation, Parallel Evaluation, Primitive Experiment Results, and Conclusion.

2. Survey of Related Work

Work related to the simulation environment described in this paper can be classified into three categories: object-oriented and parallel simulation systems, object-oriented databases, and active databases.

Object-Oriented and Parallel Simulation Systems

The features provided by the object-oriented paradigm naturally lead to the use of the object-oriented paradigm in simulation systems. To our knowledge, more than a dozen object-oriented simulation languages/systems have been developed; a survey of such systems/languages can be found in [ThMU90]. Typically, such systems/languages extend an object-oriented programming language with the necessary constructs for simulation, for instance, with "...the notion of simulation time and mechanisms for entities in the language to manipulate simulation time" [Mors90].

The problem with parallel processing of simulation systems has attracted much attention recently. A number of parallel computation models and their associated problems have been investigated [Fup90][Mors90]. The models can be classified into two categories: synchronous and asynchronous. In a synchronous, parallel simulation system, processes and events are scheduled and executed by the simulator with a global clock. Each process, however, in an asynchronous, parallel simulation system maintains its own clock, and processes and events are scheduled and executed in a fully distributed fashion. This means there exists no scheduler to synchronize the events globally. According to [Fup90], "...few simulator events occur at any single point in simulated time; therefore parallelization techniques based on lock-step execution using a global simulation clock perform poorly or require assumptions in the timing model that may compromise the fidelity of the simulation." Accordingly, "Concurrent execution of events at different points in simulated time is required, but..., thus introduces interesting synchronization problems..." Most of these synchronization problems arise from data dependencies among different processes which run at different speeds. Approaches to the synchronization problems can be in turn distinguished by two approaches: conservative and optimistic. A conservative approach prevents any synchronization problem from happening, but it degrades performance. An optimistic approach allows synchronization problems to occur, and rollbacks are often necessary once these problems are detected.

Object-Oriented Databases

In the past, several object-oriented databases have been proposed. In brief, researchers and developers have approached object-oriented database implementation along two lines: by extending the relational model (e.g., POSTGRES [Ston86] [StRo86] [RoSt87], GENESIS [BBGS86], Starburst [SCFL86], Iris [Fish87], EXODUS [GDRS86], and PROBE [DBGH86]), or by applying the ideas of object-oriented programming to permanent storage (e.g., GemStone [MSOP86] and Jasmine [MaWu86] [Wieb86]). Most of the systems in the first category have been designed to simulate semantic data models by including mechanisms such as abstract data types, procedural attributes, inheritance, union type attributes, and shared subobjects. And most of the systems in the second category extend an object-oriented programming language with persistent objects and some degree of declarative object retrieval.

Both approaches have drawbacks in processing a large number of active objects. The first approach suffers from the instability problem resulting from the separation of control and data. The second approach, on the other hand, loses the advantages provided by fact-oriented database operations.

Active Database Systems

The idea of incorporating rules into a database system as integrity constraints and triggers came about in with the early CODASYL systems in the form of *ON* conditions. More recently, the idea of combining rules and data has received much serious consideration. The term "active database" has been used frequently in referencing such databases. For example, rules have been built into POSTGRES [RoSt87]: there are no differences between constraints and triggers; all are implemented as a single rule mechanism. In addition, POSTGRES allows queries to be stored in a data field which is evaluated whenever the field is retrieved. In HiPAC [DBBC88], the concept of Event-Condition-Action (ECA) rules was proposed. When an event occurs, the condition is evaluated; if the condition is satisfied, the action is executed. It can be shown that ECA rules can be used to realize integrity constraints, alerts, and other facilities. Rules have also been included in the context of object-oriented databases. In Starburst [WiCL91], for example, rules can be used to enforce integrity constraints and to trigger consequences. In Iris [Fish87], a query can be monitored by first defining it as a

[1] This work is funded by the Office of Naval Technology, Code 227, Computer Technology Block Program through the Office of Naval Research ASEE Summer Faculty Research Program.

THIS
PAGE
IS
MISSING
IN
ORIGINAL
DOCUMENT

page 114

2. The calling object resumes execution immediately after the reply is received from the called object.

Communication between two objects is said to be *asynchronous* if the calling object continues its execution after a message is sent. The reply associated with the message is later picked up by a *receive* operation. These can be accomplished with the following method predicates.

- *m:send()*, which is true if a message *m* is sent from the object to another object. The message contains the sender, the recipient, a timestamp, the method to invoke, and a set of arguments if necessary.
- *m:receive()* which is true if a message *m* is received at the object.

Based on the above, an operation of the form *c:a(....)* implements a synchronous communication; it is equivalent to a *send* operation immediately followed by a *receive* operation. If a *send* operation is followed by a receive operation in a deterministic state, with some other operations in between them, then the communication is more or less synchronous (let us call it semi-synchronous). If a *send* operation is followed by a *receive* operation in an indeterministic state, then the communication is asynchronous.

4. Object-Oriented Simulation

As discussed in Section 2, most of the existing object-oriented simulation systems provide an object-oriented user interface so that a simulation program can be described in an easy and friendly fashion. Execution of an object-oriented simulation program can be completely sequential or fully distributed as the program specifies. As attractive as it seems, executing a simulation program as a fully distributed, object-oriented system could be inefficient due to the shortage of physical resources and the overhead associated with process management. Motivated by this, our approach is to compile an object-oriented simulation program, in which each process object is represented as a production system, into a (production) rule network. If each process object is treated as a passive object, then each node of the network corresponds to a set-oriented operation. The compiled network (or the set of operations of the network) is evaluated in parallel. Changes

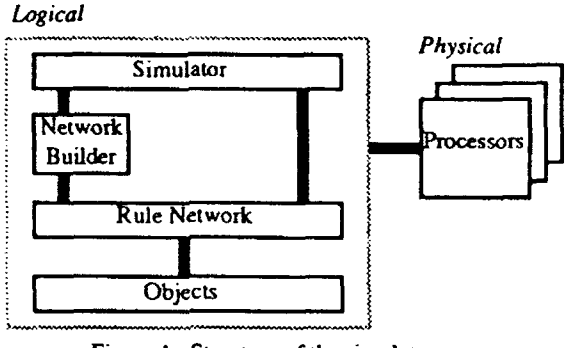

Figure 1. Structure of the simulator.

to objects are generated at the terminals of the network. Events generated by a process, if any, are expressed as new production rules and are compiled into (and removed from) the network dynamically. The overall architecture of the object-oriented simulator is shown in Figure 1. When operational, the simulator executes a loop with the following steps (see Figure 2):

Process Selection

The rule network is evaluated. At the terminals of the network, events are generated. This step basically selects those processes which have one or more productions eligible for firing based on their current states.

Production Firing

For each process selected, the actions associated with each production rule, which is ready to fire, are taken. (Note that a rule, which is ready to fire, could be an event created earlier.) Such an action could be an operation which changes the value of a (passive) object, a communication operation (send and/or receive), or an operation which produces an event (which will be executed in the future), which is converted to the form of production rules and compiled into the network by the network builder. If necessary, the value of the clock associated with the process is updated based on the operation(s). At this point, if a causality error occurs, necessary rollbacks are performed in the processes involved.

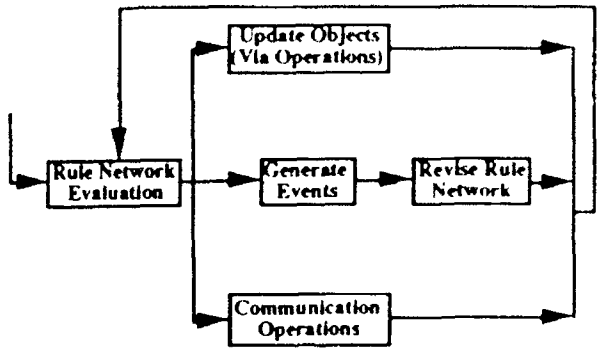

Figure 2. Logic of the simulator.

As described, this approach merges a set of process objects into a (much) smaller set of operation processes. Simulation can be performed synchronously or asynchronously. Asynchronous simulation is more complicated as different versions of the same object must be considered in making a decision. This will be reported in a separate paper. In the synchronous mode, a global clock is employed so that every process object is synchronized with respect to the global clock. The network can be evaluated continuously every clock cycle or discretely according to the events produced. It requires, however, that the states of each object be recorded and rollbacks be performed whenever causality errors are detected.

Since the object model allows objects to be shared among different processes (although they are accessed through messages), *serializability* [BeHG87] must be maintained all the time. This means that the effects created by multiple processes which are executed concurrently should be the same as those created by a (any) serial schedule among the processes. To assure this, the design employs the two-phase locking protocol, which requires all objects accessed by a process be locked before accessed, that all locks be acquired before any unlock, and that all objects be unlocked before the process terminates. Clearly, two-phase locking cannot be implemented at the method level, because two consecutive method calls can violate the two-phase requirement. Consequently, it is required that each method lock any object it may access but not to subsequently unlock it. The list of locked objects should be returned to the calling process so that it can unlock the locked objects before terminating.

5. Rule Processing

In general, the processing of production rules or integrity constraints can create serious performance bottlenecks when a large volume of facts and rules are integrated. Since multiple instances of the same class share the same copy of production rules, it is useful to compile a set of rules into one system in which some set-oriented operations can be employed to process the data (treated as sets) collectively. Furthermore, given a set of rules, it is fruitful to merge those expressions that are common to more than one rule so that duplicated effort can be avoided. A network approach is taken for this purpose. This approach is similar to the RETE algorithm ([ACAR86][Forg82]), but is more general in the treatment of logical formulas. Although integrity constraints and production rules are treated slightly differently, both are processed based on a network that is compiled from a set of logical formulas.

Processing Integrity Constraints

Given a set of constraints $\{f_1 \rightarrow r_1, \ldots, f_n \rightarrow r_n\}$, at the outset, each constraint $f_i \rightarrow r_i$ is converted into the form $f_i \wedge \neg r_i$ (i.e., the negation of the original rule). All the converted rules are subsequently compiled into a network (see below) in which each rule corresponds to a terminal at the bottom of the network, and there is no violation of the rule if no result can "flow" out from that terminal.

THIS
PAGE
IS
MISSING
IN
ORIGINAL
DOCUMENT

page 116

$$cost\ (RN) = C_B + \sum_{i=1}^{l} |nj_i| + \sum_{j=1}^{m} |ns_j|$$

$$= C_B + \sum_{i=1}^{l} [a_i * f\ (e\ (nj_i, 1)) * f\ (e\ (nj_i, 2))]$$

$$+ \sum_{j=1}^{m} [b_j * f\ (e\ (ns_j))]$$

where C_B is the sum of the cardinalities of the base relations which are initially read for joins or selections. Since MGFs are counted only once, the sharing of information in the relational network are reflected in the above cost function. Once a network is built, it can be evaluated incrementally. Specifically, each operation is evaluated once based on the initial state of the system. In the meantime, for each operation, the results are stored. Subsequently, as the state of the system is changed, only those rules which are affected by a changed fact need to be evaluated during each iteration. When an update of the database is made, operations are performed from the bottom of the network. First, the updated fact is matched against the MGPs. Only those MGPs having the same head as the updated fact and whose arguments can be unified by the arguments of the updated fact are activated. After the appropriate MGPs are activated, the operations connected to the activated MGPs are activated. For each activated operation node, the content of the stored result is changed according to the change(s) in its input relations. If the update is an addition of a new fact, a new tuple of values may be added after a select operation. If the update is a deletion, the tuple corresponding to the deleted fact may be deleted from the result stored in the operation node after a select operation. Similarly, tuples may be deleted from the result of a join operation if some tuples are deleted from its inputs; and tuples may be added into the result of a join operation if some tuples are added into its inputs. A modification of an existing fact can be handled by first deleting the old fact followed by adding the new fact. After all the related operations to an update are performed, the final changes obtained at each terminal node are applied to the associated action(s).

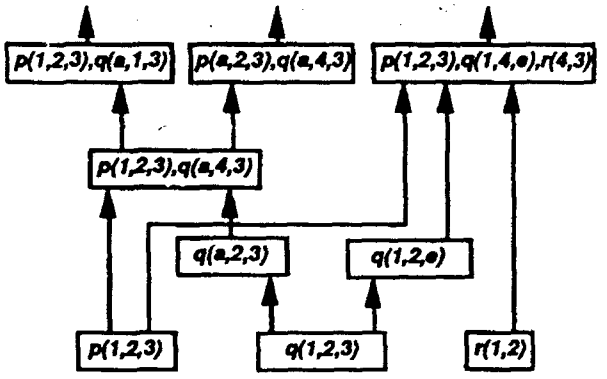

Figure 3. A rule network.

6. Incremental Network Compilation

Let N be a rule network corresponding to the set of production rules $R = \{r_1,...,r_t\}$ and let $p = p_1 \wedge p_2 \wedge ... \wedge p_n$ be the condition part of a new production rule r'. Also let $N_n = \{n_1,...,n_s\}$ be the set of nodes of N and the function, f, map each n_i, $1 \le i \le s$, to its corresponding conjunctive formula. A simulator requires that N be revised with minimal effort to a new network N' which corresponds to the augmented rule set $R' = \{r_1,...,r_{t'}\}$.

Instead of recompiling the augmented rule set from scratch, it is desirable to incrementally connect the new rule into the existing network. To realize this, a *cover* of the new rule, r', with respect to N is defined to be any subset of $\{n_1,...,n_s\}$ from which r' can be derived. The *cost* of a cover is defined to be the sum of the costs associated with the nodes and the arcs which have to be added into the network so that r' can be computed from the nodes of the cover. Among all the covers, the one with the smallest cost is selected to implement r' with the current network. The following procedure can be followed to identify all the possible covers for r'.

Identifying Covers for A New Rule

input: N and r' as described in the above

output: C, the set of all the covers for r' with respect to N.

Step 1

Let $C = \phi$, $T = \{\{n_i\} \mid n_i \in N_n\}$

Step 2

Find u and v of T where $f(u) \wedge f(v)$ is more general than a subformula of r' and $\{u,v\}$ is not in T or C. If at this point $u \wedge v = r'$, $\{u, v\}$ is a cover for r'; therefore $C = C \cup \{u, v\}$. Otherwise, let $T = T \cup \{u, v\}$ Repeat this step until no more of u and v pairs can be found.

□

7. Parallel Evaluation of a Rule Network

The following approaches can be applied in order to evaluate a rule network, depending on how logical objects are packed into physical objects:

Class-Level Parallelism

In this approach, each node of the network is implemented as a physical object, where each terminal node is a class object and each internal node is an operation object. The network is evaluated as an active network which operates in a pipelined fashion. Specifically, each operation object retrieves inputs from its input object(s) and produces the outputs, which are available to the operation object(s) at the next higher level. Unlike operation nodes, each class object functions as a data store from which data can be retrieved by operation objects.

Set-Level Parallelism

In this approach, each terminal node is implemented as a set of objects, in which each object corresponds to a subset of a class. The network is transformed into an equivalent network in which each terminal node corresponds to a subclass object. The transformation can be done in a straight-forward fashion based on the following principles:

1. $(R = R_1 \cup R_2) \wedge (S = S_1 \cup S_2) \Rightarrow R\ |x|\ S = (R_1\ |x|\ S_1) \cup (R_1\ |x|\ S_2) \cup (R_2\ |x|\ S_1) \cup (R_2\ |x|\ S_2)$

2. $(R = R_1 \cup R_2) \Rightarrow select_F(R) = select_F(R_1) \cup select_F(R_2)$

Clearly, this approach can achieve a higher degree of parallelism; however it is more complicated to implement. In addition, the number of operator objects can grow exponentially as each class is split into smaller and smaller subsets.

8. Primitive Experiment Results

Some primitive experiments have been performed on an ENCORE parallel computer (with four processors) to compare the performance of different approaches for a simple scenario. The scenario consists of a number of battle divisions comprising two sides, blue and red. The divisions are initially located on the boundaries of a battlefield which is modeled by a square of grid tiles. The scenario is set up so that all the red divisions are distributed on the east border of the battlefield and the blue divisions are distributed on the west side. Once initiated, the blue divisions march to the west and the red divisions march to the east. Throughout the simulation each division is characterized by its strength, speed, direction of movement, and its location. When two divisions of opposite sides encounter each other, the strength of the weaker is reduced to zero, and the strength of the stronger is reduced by that of the weaker. On the other hand, when two divisions of the same side meet with each other, they are merged into a larger division. In any case, the number of divisions in each grid tile cannot exceed two.

The scenario was simulated using four approaches: *sequential* (*S*), *parallel synchronous* (*PS*), *asynchronous* (*A*), and the *network approach* (*N*) as